CrystalNet: Faithfully Emulating Large Production Networks

Hongqiang Harry Liu[★], Yibo Zhu[★], Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes,
Andrey Rybalchenko, Guohan Lu, Lihua Yuan
Microsoft
{harliu,yibzh,padhye,jiacao,srita,nlopes,rybal,gulv,lyuan}@microsoft.com

ABSTRACT

Network reliability is critical for large clouds and online service providers like Microsoft. Our network is large, heterogeneous, complex and undergoes constant churns. In such an environment even small issues triggered by device failures, buggy device software, configuration errors, unproven management tools and unavoidable human errors can quickly cause large outages. A promising way to minimize such network outages is to proactively validate all network operations in a high-fidelity network emulator, before they are carried out in production. To this end, we present *CrystalNet*, a cloud-scale, high-fidelity network emulator. It runs real network device firmwares in a network of containers and virtual machines, loaded with production configurations. Network engineers can use the same management tools and methods to interact with the emulated network as they do with a production network. *CrystalNet* can handle heterogeneous device firmwares and can scale to emulate thousands of network devices in a matter of minutes. To reduce resource consumption, it carefully selects a boundary of emulations, while ensuring correctness of propagation of network changes. Microsoft's network engineers use *CrystalNet* on a daily basis to test planned network operations. Our experience shows that *CrystalNet* enables operators to detect many issues that could trigger significant outages.

CCS CONCEPTS

•Networks → Network reliability;

KEYWORDS

Network, Reliability, Emulation, Verification

1 INTRODUCTION

CrystalNet is a high-fidelity, cloud-scale network emulator in daily use at Microsoft. We built *CrystalNet* to help our engineers in their quest to improve the overall reliability of our networking infrastructure. A reliable and performant networking fabric is critical to meet the availability SLAs we promise to our customers.

It is notoriously challenging to run large networks like ours in a reliable manner [11, 13, 15, 31]. Our network consists of tens of thousands of devices, sourced from numerous vendors, and deployed across the globe. These devices run complex (and hence bug-prone) routing software, controlled by complex (and hence bug-prone) configurations. Furthermore, churn is ever-present in our network: apart from occasional hardware failures, upgrades, new deployments and other changes are always ongoing.

The key problem is that in such a large and complex environment, even small changes or failures can have unforeseen and near-disastrous consequences [16]. Worse yet, there are few tools at our disposal to proactively gauge the impact of failures, bugs or planned changes in such networks.

Small hardware testbeds [1, 2] are used to unit-test or stress-test new network devices before they are added to the network. While useful, these cannot reveal problems that arise from complex interactions in a large topology.

Network verification tools such as Batfish [13] ingest topology and configuration files, and compute forwarding tables by simulating the routing protocols. These forwarding tables can be analyzed to answer a variety of reachability questions. However, Batfish cannot account for bugs in routing software. Nor can it account for subtle interoperability issues that result from slight differences in different vendor's implementation of the same routing protocol. In other words, Batfish is not "bug compatible" with production network. In our network nearly 36% of the problems are caused by such software errors (Table 1). Note that there is no way to make Batfish bug compatible – often, the bugs are unknown to the device vendors themselves until they manifest under certain conditions. Also, Batfish presents a very different workflow to the operators of the network. This means it is not suitable for preventing human errors, which are responsible for a non-negligible 6% of the outages in our network.

What we needed was a large scale, high-fidelity network emulator that would allow us to accurately validate any planned changes, or gauge the impact of various failure scenarios. Small-scale network emulators such as MiniNet [18] or GNS3 [3] are useful, but have many deficiencies (see §10), and do not

[★]: co-primary authors

SOSP '17, Shanghai, China

scale to the level required to emulate large public cloud networks.

To address this gap, we built *CrystalNet*, a highly scalable, and high-fidelity network emulator. High fidelity means that the emulator accurately mimics the behavior of the production network, especially in the control plane. Further, it allows the operators to use the exact same tools and workflows that they would on a production network.

We do not claim full fidelity - that requires creating a complete replica of the production network, which is infeasible. Thus, it is not our goal to faithfully emulate the network dataplane (latency, link bandwidth, traffic volume etc.). Our focus is on the control plane.

To accurately mimic the control plane, *CrystalNet* runs real network device firmwares in virtualized sandboxes (*e.g.,* containers and virtual machines). Such VMs or containers are available from most major router vendors. We inter-connect the device sandboxes with virtual links to mimic the real topology. It loads real configurations into the emulated devices, and injects real routing states into the emulated network.

Operators can interact (i.e. change, upgrade, or monitor) with the emulated network using the same tools and scripts they use to interact with the production network. They can inject packets in the emulated network and monitor their paths. *CrystalNet* can even be extended to include real hardware in the emulated network.

Our network engineers use *CrystalNet* on a daily basis. They have caught several errors in their upgrade plans which would have been impossible to catch without an emulator like *CrystalNet*. We report some of their experiences in §7.

CrystalNet is scalable and cost-effective. To emulate a network of 5,000 devices, we need just a few minutes and 500 VMs (4 cores, 8GB RAM). Such VMs retail for USD 0.20/hour each, so the total cost of emulating such a large network with *CrystalNet* is USD 100/hour. This is miniscule compared to the cost of a network outage.

Three key features allow *CrystalNet* to scalably emulate large, heterogeneous networks, which are also our major contributions in this paper. First, *CrystalNet* is designed to run from ground-up in public cloud. If necessary, *CrystalNet* can even simultaneously use multiple public and private clouds. This allows *CrystalNet* to scale to levels well beyond those possible with MiniNet and GNS3. Since VM failures are likely to happen in any large-scale deployment, *CrystalNet* allows saving and restoring emulation state, and quick incremental changes to the emulation.

Second, *CrystalNet* can accommodate a diverse range of router software images from our vendors. The router images are either standalone VMs or in form of Docker containers. To accommodate and manage them uniformly, we mock-up physical network with homogeneous containers and run heterogeneous device sandboxes on top of the containers' network namespace. *CrystalNet* also allows our engineers to access the routers in a standard manner via Telnet or SSH. *CrystalNet* can also include on-premise hardware devices in the emulated network in a transparent manner. This requires careful traversal of NATs and firewalls in the path.

Third, *CrystalNet* accurately mock-up external networks transparently to emulated networks. An emulated network has to have a boundary, beyond which there are no emulated devices. Apart from resource constraints (one cannot emulate the whole Internet), the fact is that we cannot obtain the configurations and device firmware from devices that are outside our administrative control (*e.g.,* our upstream ISP). We use lightweight passive agent that mimics the announcements emulated devices would receive from beyond the boundary. Since the agents do not respond to dynamics in the emulated network, we ensure the correctness of the results by identifying and searching for a *safe* boundary (§5). Computing the right boundary can also save resources: indeed, it can cut the cost of emulation by 94-96% while maintaining high fidelity (§8.4).

Before describing *CrystalNet* in more details, we first discuss the outages in our network over the last two years.

2 MOTIVATION

Table 1 shows a summary of O(100) network incidents in our network and their root causes for the past two years. The categories are broad, and somewhat loosely defined; what matters are the details of individual scenarios, as below.

Software bugs This category includes incidents caused by issues in device firmware, and bugs in our own network management tools; although, most incidents are due to bugs in device firmware.

Examples of bugs in our own automation tools include an unhandled exception that caused a tool to shut down a router instead of a single BGP session.

Device software issues come in many forms. Some are outright bugs: for example, new router firmware from a vendor erroneously stopped announcing certain IP prefixes. In another case, ARP refreshing failed when peering configuration was changed.

Another set of problems arise out of ambiguity, rather than bugs. Different versions of the network devices from the same vendor usually have slightly different configuration definitions. For instance, a vendor changed the format of ACLs in the new release, but neglected to document the change clearly. As a result, the old configuration files were processed incorrectly by switches running the new firmware.

Devices often exhibit vendor-dependent behavior in the implementation of otherwise standard protocols/features, e.g.,

Root Cause	Proportion	Examples	*CrystalNet* Coverage	Verification Coverage
Software Bugs	36%	bugs in routers, middleboxes, management tools	✓	X
Config. Bugs	27%	wrong ACL policies, traffic black holes, route leaking	✓	✓
Human Errors	6%	mis-typing, unexpected design flaws	✓	X
Hardware Failures	29%	ASIC driver failures, silent packet drops, fiber cuts, power failures	X	X
Unidentified	2%	transient failures	X	X

Table 1: Root causes of O(100) significant and customer-impacting incidents in our network (2015 - 2017).

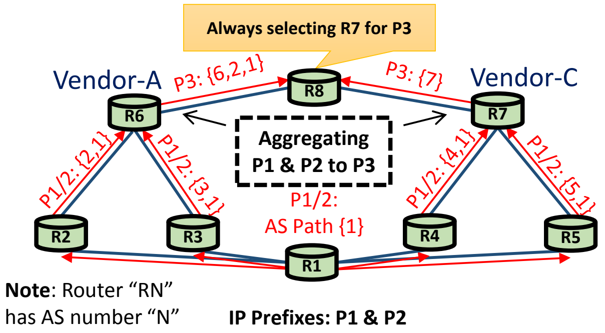


Figure 1: Traffic load imbalance caused by vendor specific behaviors in IP aggregation.

how to select BGP paths for aggregated IP prefixes, or how to process routes after FIB is full, etc. Such corner cases are often not well documented. For example, Figure 1 shows a simplified version of a problem we saw in production. IP prefixes P1 and P2 belong to router R1 with AS number "1". When higher layer routers R6 and R7 get the announcements of these two prefixes, they want to aggregate them into a single one (P3). However, R6 and R7 are from different vendors, and they have different behaviors to select the AS path of P3: R6 learns different paths for P1 and P2 from R2 (with AS path {2, 1}) and R3 (with AS path {3, 1}) and it selects one of them and appends its own AS number before announcing P3 to R8 ({6, 2, 1} in this example); R7 faces a similar situation, but it does not select any paths and only puts its AS number as the AS path in the announcement of P3 to R8. As a result, R8 always prefers to send packets for P3 to R7 because it thinks R7 has a lower cost, causing sever traffic imbalance.

Sometimes, different system components that perform correctly in individual capacity, do not interact well, especially after a change. For instance, a software load balancer owned a /16 IP prefix. However, it was asked to release some IP blocks in the prefix and give them to other load balancers. It then broke the /16 IP prefixes into 256 × /24 IP blocks and announced the blocks (about 100) that it held. However, a router connected to the load balancer was short of FIB space and dropped many of these announcements, causing traffic black holes.

Note that these errors escaped the fairly rigorous unit testing done by our vendors as well as our own pre-certification checks. While more rigorous unit testing is always helpful, it is impossible to cover the vast range of possible inputs and conditions that occur in production environment. Full-fledged integration testing would be impractically resource intensive – unless one used a high fidelity emulator like *CrystalNet*. We do not claim that *CrystalNet* can uncover all bugs – only that by letting operators test router firmwares, tools and planned changes in a high-fidelity emulation would reduce the possibility of such bugs impacting production networks.

We note once again that network verification systems [11–13, 15, 23, 30] cannot account for the impact of such bugs, since they rely on analyzing configurations, and assume ideal, bug-free behavior from network components. One may think that the systems can be updated to model the bugs – but many of the bugs are "unknown" until they manifest themselves in production networks.[1] Moreover, such systems are even less effective when the network has components like software load balancers, whose behavior is "baked" into custom software, rather than driven by configurations and governed by standards. One can never fully and accurately model the behavior of such components.

Configuration bugs: Network configurations are not just for controlling behavior of routing protocols – their design must also account for issues like forwarding table capacity, CPU load of devices, IPv4 shortage and reuse, security and access control, etc. Taken together, this makes our network configuration policies quite complicated. As a result, 27% of outages result from what can be termed as configuration errors, such as missing or incorrect ACLs violate security, overlapping IP assignments, incorrect AS number etc.

Our devices are initially configured automatically, using a configuration generator similar to [9, 28]. Most of the incidents in this category were triggered due to ad-hoc changes to configurations during failure mitigation or planned updates. By testing such changes with *CrystalNet*, the possibility of such errors impacting production networks can be reduced.

Human errors We define "human errors" as those manual actions that clearly mismatch their intention, resulting in an error of some kind. e.g. mistyping "deny 10.0.0.0/20" as

[1] And they are typically fixed soon afterwards, so there is even less value in modeling them.

"deny 10.0.0.0/2". Human errors surprisingly cause a non-negligible portion (6%) of the incidents. One might argue that this is due to carelessness and cannot be remedied. However, after conversations with experienced operators, we found a more important systematic reason is that operators do not have a good environment to test their plans and practice their operations with actual device command interfaces. *CrystalNet* can provide such an environment. Network verification systems like Batfish present a different workflow than what the operators would carry out in practice, and hence cannot reduce the occurrence of such errors.

Summary The analysis of these incidents underscores the fact that numerous different types of bugs and errors can affect large, complex networks. Testing and planning with a high-fidelity network emulator like *CrystalNet* can catch many of these bugs and errors before they disrupt production networks; while traditional network verification systems offer much more limited succor.

3 *CRYSTALNET* DESIGN OVERVIEW

In this section, we describe the design goals, the overall architecture and the programming interfaces of *CrystalNet*.

3.1 Design goals

The ultimate goal of *CrystalNet* is to provide *high fidelity* network emulations to operators. To meet this goal, *CrystalNet* has three key properties:

Ability to scale out using public clouds: Resources required for faithful emulation of large networks are well beyond the capacity of a single server, or even a small cluster of servers. For example, a single Microsoft datacenter can consist of thousands of routers. Emulating each router requires non-trivial amounts of CPU, RAM etc. Even more resources are needed if we consider middleboxes and inter-datacenter scenarios. Computing resources at this scale are only available from public cloud providers in form of VMs. Thus, to ensure that there is no an upper limit on the scale of emulated networks, *CrystalNet* must be able to run in a distributed manner, over a large number of VMs in a public cloud environment. Everything should easily scale out – e.g. to double the emulated network size, the operators simply need to allocate twice the computing resources.

Ability to transparently mock up physical networks: A switch OS assumes it runs on top of a physical switch that has multiple network interfaces and connects to neighboring devices. Management tools assume each network device can be visited via an IP address with Telnet or SSH. *CrystalNet* must create virtual network interfaces, virtual links and virtual management networks that are transparent to switch OS and management tools, so that the latter can work without modifications.

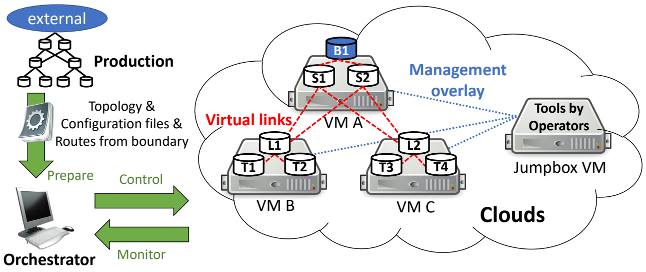


Figure 2: *CrystalNet* architecture. This shows an emulated Clos topology of eight switches running on three VMs.

Ability to transparently mock up external networks: An emulated network always has a boundary – we cannot emulate the whole Internet. This is not just a resource issue; the key problem is that operators cannot obtain OS images or configurations of devices outside their management domain. *CrystalNet* must accept the fact that boundary exists, and ensure high fidelity even though devices outside the boundary are not emulated.

In addition, we also desire properties such as failure resilience and cost efficiency. Next, we describe how the design of *CrystalNet* achieves these goals.

3.2 Architecture

Figure 2 shows the high-level architecture of *CrystalNet*. The orchestrator is the "brain" of *CrystalNet*. It reads the information of production networks, provisions VMs (*e.g.,* VM A) on clouds, starts device virtualization sandboxes (*e.g.,* T1) in the VMs, creates virtual interfaces inside the sandboxes, builds overlay networks among the sandboxes, and introduces some external device sandboxes (*e.g.,* B1) to emulate external networks. With aggressive batching and parallelism, the orchestrator runs on a single commodity server and easily handles O(1000) VMs.

CrystalNet is easy to scale-out. The overlay network ensures that that emulation can run on top of any VM clusters (with sufficient resources) without any modifications.

The emulated network in *CrystalNet* is transparent. Each device's network namespace has the same Ethernet interfaces as in the real hardware; the interfaces are connected to remote ends via virtual links which transfer Ethernet packets just like real physical links; and the topology of the overlay network is identical with the real network it emulates (§4). Therefore, the device firmware cannot distinguish whether it is running inside a sandbox or on a real device. In addition, *CrystalNet* creates a management overlay network which connects all devices and jumpbox VMs. Management tools can run inside the jumpboxes and visit the devices in the same way as in production.

The emulation boundary of *CrystalNet* is transparent. The external devices for emulating external networks provide the same routing information as in real networks. Also, as discussed in §5, the boundary is carefully selected, so that the state of the emulated network is identical to real networks even if the emulated network is under churn.

The emulated network is highly available, because VMs are independently set up – a VM does not need to know the setup of any other VMs. Thus, the orchestrator can easily detect and restart a failed VM.

CrystalNet achieves the cost efficiency by putting multiple devices on each VM, and picking the right devices to emulate rather than blindly emulate the entire network (§5.2).

3.3 *CrystalNet* APIs

The orchestrator exposes an API that operators use to configure, create, and delete emulations, and also to run various tests and observe network state for validation. The API, shown in Table 2, is inspired by the validation workflows which network operators desired to run.

Figure 3 illustrates the typical workflow of a network configuration update. First, `Prepare` is called to take a snapshot of the production environment, spawn VMs and feed those as the input into `Mockup`. `Prepare` includes functionality to get the necessary topology information, device configurations, and boundary route announcements (see §5), and VM planning based on topology. `Mockup` creates the virtual network topology (§4) and the emulation boundary (§5), and starts the emulated device software.

After `Mockup`, *CrystalNet* is ready for testing the update steps. At each step, operators can choose to apply significant changes like booting a new device OS or updating the whole configuration with `Reload`, or use existing tools for incremental changes via the management plane (§4).

Next, the operators can pull the emulation state (e.g. routing tables at each device) using monitoring APIs, as well as their own tools, to check whether the changes they made had the intended effect. *CrystalNet* also supports packet-level telemetry [32] for this purpose. Operators specify the packets to be injected and *CrystalNet* injects them with a pre-defined signature. All emulated devices capture all seen packets, filter and dump traces based on the signature. These traces can be used for analyzing network behavior.

With the ability to obtain routing tables, packet traces and the ability to login to emulated devices and check device status (see Table 2), operators using *CrystalNet* can validate an emulated network using their preferred methodologies[2], e.g. injecting test traffic, verifying routing tables with *reactive* data plane verification tools [22], etc. If the results are as expected, operator can move onto the next step. Otherwise, operators revert current update with `Reload`, fix the bugs and try again. This process repeats until all update steps are validated. In the end, `Destroy` is called to release VMs.

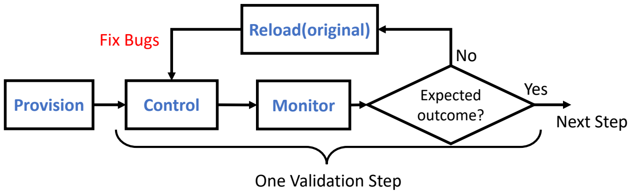


Figure 3: A typical network update validation workflow. *CrystalNet* APIs cover the parts in blue and bold. Rest of the workflow is operator-specific.

API	Description
Provision functions	
Prepare	Gather information for *Mockup*, spawn VMs.
Mockup	Create the emulation based on Prepare output.
Clear	Clean up everything on VMs.
Destroy	Erase all Prepare output, including the VMs.
Control functions	
Reload	Reboot devices with specified software versions and configurations.
Connect	Connect two interfaces.
Disconnect	Disconnect two interfaces.
InjectPackets	Inject packets with a specified header from a specified device & port, at given frequency in given amount of time.
Monitor functions	
PullStates	Pull common states from the device software, *e.g.,* FIB, RIB, CPU and memory usage, etc.
PullConfig	Back up the current configurations for rollback.
PullPackets	Pull the packet traces to local, and (optional) compute packet paths and counters (optional) clean traces after pulling.
Management plane: complementary to Control and Monitor	
IP Access	Enable existing tools to send commands or pull outputs. This is not a typical API – see §4.2.

Table 2: Selected *CrystalNet* APIs.

CrystalNet also offers several helper APIs such as `List` all emulated devices, `Login` to a device, etc. We omit the details.

The key part of *CrystalNet* is to `Mockup` a high-fidelity environment that supports this unified API set and is cost-effective. We discuss it in the next two sections.

4 MOCK UP PHYSICAL NETWORKS

4.1 Heterogeneous network devices

CrystalNet supports various OSes and software running on network devices. We focus on switches in our datacenter and WAN networks, which include three of the largest switch vendors (referred as *CTNR-A*, *VM-B* and *VM-A*), and an

[2] Designing automated testing methodologies using *CrystalNet* is an important, but orthogonal research topic.

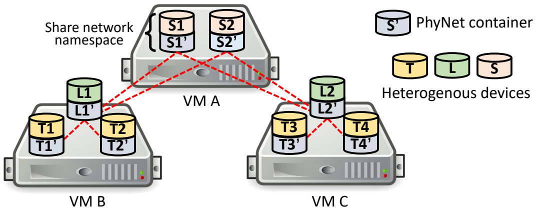


Figure 4: PhyNet containers in *CrystalNet* decouple the interfaces' management and facility tools from the device software.

open source switch OS (*CTNR-B*). *CrystalNet* is designed to run transparently with these heterogeneous software systems, be extensible to other device software, and provide unified APIs (§3.3) for users.

CrystalNet chooses containers as the basic format for isolating devices. Containers isolate the runtime library with less overhead than VM, run well inside VMs on clouds, and, more importantly, isolate virtual interfaces of multiple devices to avoid naming conflicts. We use *Docker* engine to manage containers. We address challenges of running heterogeneous software, as explained below.

A unified layer for connections and tools. *CrystalNet* APIs must work for all devices we want to emulate. However, the heterogeneous device software is packed into different blackbox images by vendors. It is daunting, sometimes infeasible, to re-implement the APIs for each device and ensure consistent behavior. Another engineering challenge is that most containerized switch OS must boot with interfaces already present, while virtual interfaces can only be put into a container after the Docker container boots.[3]

To address this, we design a unified layer of Physical Network, or *PhyNet* containers (Figure 4), whose runtime binaries are decoupled from the devices being tested. This layer of containers hold all the virtual interfaces and are connected as the target topology. We place common tools, like Tcpdump, packet injection and pulling scripts, in PhyNet containers. Most *CrystalNet* APIs are then implemented for these PhyNet containers, instead of being re-implemented for each device.[4] Later, we boot the actual device software with the corresponding network namespace. Thus, the device software runs without any code changes – just like in the real life, they start with the physical interfaces already existing. Even if the software reboots or crashes, the virtual interfaces and links remain. The overhead of running PhyNet containers, which exists only to hold network namespaces, is negligible.

VM-based devices. While some vendors offer containerized images, others, like *VM-B* and *VM-A*, offer only VM images of their switch software. We cannot run VM-based device image directly on clouds, because public clouds cannot attach hundreds of virtual interfaces to a VM. In addition, we need to connect these VM-based devices with other containers, and maintain the PhyNet container layer.

Our solution is to pack the VM images, a KVM hypervisor binary, and a script that spawns the device VM, into a container image. In other words, we run the device VM inside a container on the cloud VMs. This requires *nested VM* feature on clouds. This feature is available on Microsoft Azure, as well as some other public clouds. In absence of this feature, *CrystalNet* can provision bare-metal servers for VM-based devices instead.

Real hardware. Finally, *CrystalNet* also allows operators to connect real hardware into the emulated topology. For example, *CrystalNet* can mock up a full network with one or more devices replaced by the real hardware. This allows us to test the hardware behavior in a much more realistic environment than the traditional stand-alone testing. Each real hardware switch is connected to a "fanout" switch. The "fanout" switch tunnels each port to a virtual interface on a server. These virtual interfaces are managed by a PhyNet container and are bridged with virtual links (see § 4.2) connecting the *CrystalNet* overlay.

By introducing PhyNet containers, *CrystalNet* is able to treat devices identically, regardless of whether they run in containers, VMs or as true physical devices, from the management viewpoint.

4.2 Network links

There are two types of virtual links in *CrystalNet*, one for the data plane and the other for the management plane.

Data plane. The virtual data plane links should be seen as Ethernet links by the devices and should be isolated from one another. Furthermore, the virtual links must be able to go through underlying networks, including the cloud provider's network, and the Internet. The ability to travel over Internet in a seamless manner is necessary to allow the emulation to span multiple public clouds, and to allow cloud-based emulations to connect to one or more physical devices.

We choose VXLAN over other tunneling protocols (*e.g.*, GRE) because it meets our goal best – it emulates an Ethernet link, and the outer header (UDP) allows us to connect across any IP network, including the wide area Internet. We can even cross NATs and load balancers, since most of them support UDP.[5]

As shown in Figure 5, each device interface is a member of a *veth* pair [18], with the other side plugged into a bridge.

[3] These containers are originally designed for deployment on real hardware and in host network namespace, while *CrystalNet* runs them in non-host, isolated namespaces.

[4] Only `PullConfig` and `PullStates` are notably different across devices. This is unavoidable.

[5] We use standard UDP hole punching techniques [14].

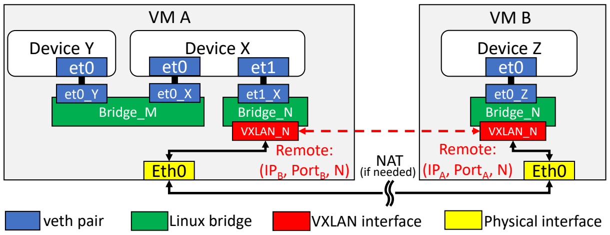


Figure 5: The design of virtual data links. Device X's $et0$ is connected to Y's $et0$. X's $et1$ is connected to Z's $et0$. N is a VXLAN ID. The devices shown are PhyNet containers.

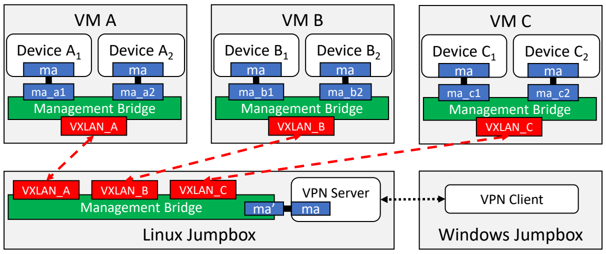


Figure 6: The architecture of management plane. Physical interfaces are omitted for brevity.

Each bridge also has a VXLAN tunnel interface (if the remote device is on another VM), or another local *veth* interface. This is transparent to the device containers. We isolate each virtual link by assigning a unique VXLAN ID to each link. Orchestrator ensures that there is no ID collision on the same VM.

Management plane. Through the years, operators have developed tools based on direct IP access to devices through the management plane which is an out-of-band channel just for management. *CrystalNet* provisions this management plane automatically (Figure 6). Operators can run their management tools *without any modifications*, perform incremental configuration changes with the tools, and pull device state, just like in production environments.

CrystalNet deploys a Linux jumpbox[6], and connects all emulated devices together. However, one cannot simply connect all management interfaces in a full L2 mesh - this would cause the notorious L2 storm in such an overlay. Instead, we build a tree structure – each VM sets up a bridge and connects to the Linux jumpbox via VXLAN tunnels. All emulated devices connect to the bridge of the local VM. Other jumpboxes, like a Windows-based jumpbox, connect to the Linux jumpbox via VPN. Finally, the Linux jumpbox runs a DNS server for the management IPs of the devices.

5 MOCK UP EMULATION BOUNDARY

Any emulated or simulated network must have a boundary – for example, when we simulate one or more of our data centers, we stop at the point where they connect to the wide area Internet. Going beyond this boundary is infeasible, not just because we lack resources, but also because we cannot obtain configuration or other information for the devices that are not under our control. However, the routing messages from external networks, and the reactions of external devices to the dynamics inside the emulated network, are essential to ensure the correctness of the emulation.

The key to solving this dilemma is based on the observation that most production networks do not blindly flood or accept route updates. Upon dynamics in a particular location, there are policies or protocols limiting the range of impact. If we find the stopping boundary of the impact, we can safely assume that the network outside the boundary remains static during the period when operators validate the network inside the boundary. In the rest of the section, we discuss this concept in more details.

5.1 Static emulation boundary

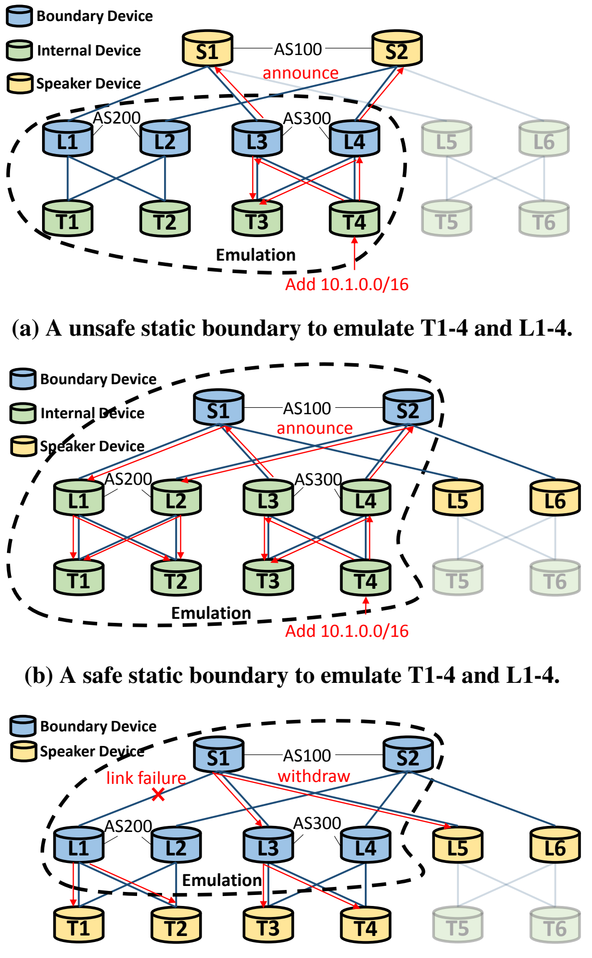
In *CrystalNet*, we define an *emulated device* as a device running actual device firmware and configurations from production. For example, in Figure 7a, T1-4 and L1-4 are all emulated devices. Furthermore, we call T1-4 *internal devices* since their neighbors are all emulated devices, and L1-4 *boundary devices* since they have neighbors outside the boundary. We call the external devices that are directly connected to boundary devices as *speaker devices*. Other external devices are excluded from emulation. For instance, in Figure 7a, S1-2 are speaker devices and are connected to emulated devices, while T5-6 and L5-6 are not emulated.

In *CrystalNet*, speaker devices do not run the actual device firmware or configurations from production. Instead, they run simple software and perform just two functions. First, they keep the link(s) and the routing session(s) alive with boundary devices, so that the existence of the boundary is transparent to emulated devices. Second, they are fully programmable for sending out arbitrary routing messages.

Static speaker devices In *CrystalNet*, we design speaker devices to be *static*, *i.e.,* the speaker devices do not react to any routing messages from boundary devices. Instead, during `Prepare`, *CrystalNet* installs the routing messages to be sent by each boundary device. After `Mockup`, the speaker devices announce these messages. By making the speaker devices static, we avoid any assumptions on the behavior of external devices.

An alternative is to design a dynamic boundary with a network simulator which runs a canonical implementation of routing protocols. This simulator can compute how each

[6] Use of jumpboxes is common in production networks.



(a) A unsafe static boundary to emulate T1-4 and L1-4.

(b) A safe static boundary to emulate T1-4 and L1-4.

(c) A safe static boundary to emulate L1-4 but not T1-4.

Figure 7: Examples of unsafe and safe static boundaries based on the devices need to be emulated in a BGP datacenter network.

speaker device should react in real time. However, we did not choose this option for two reasons. First, we typically do not have access to external devices' policies or configurations, so that we cannot fully simulate them. Second, the canonical routing protocol implementation is likely to have bugs of its own, which may affect the correctness of the overall emulation.

Safe static boundary The use of static speaker devices raises a concern: is the emulation still correct when operators apply changes on the emulated devices? In other words, in the real network, will the devices represented by speaker devices react to the changes, and become inconsistent with our static design?

This concern is valid. For example, in Figure 7a, on a datacenter network which uses Border Gateway Protocol (BGP), we run T1-4 and L1-4 as emulated devices and S1-2 as speaker devices. If T4 gets a new IP prefix (10.1.0.0/16), the announcements of this prefix will stop at S1-2 in emulation. However, in real networks, S1-2 would propagate this prefix to L1-2! Given this potential inconsistency, we call the boundary in Figure 7a *unsafe*.

We define a *safe static boundary* as a collection of boundary devices that can guarantee the consistency between the emulation and the real network, even when the topology and/or configurations of emulated devices change. For example, Figure 7b has a safe boundary: S1 and S2. Announcements of the new IP prefix can reach L1-2 and T1-2, since S1 and S2 are emulated devices this time. We now prove the safeness of S1-2 as a boundary under arbitrary route and/or topology updates on T1-4 and L1-4.

5.2 Identifying and searching safe static boundaries

During `Prepare`, *CrystalNet* takes "must-have devices", the devices required by operators to emulate, as input. It then finds a safe static boundary inside which all must-have devices are emulated. In this section, we present sufficient conditions to judge the safeness of a given boundary on various networks, and a heuristic for finding a safe static boundary inside datacenter networks running BGP.

BGP networks. BGP is not only the *de facto* routing protocol between autonomous systems (ASes) in the Internet, but also widely used inside datacenter networks [4, 5, 27]. As a variant of distance vector protocols, BGP lets each router report which IP prefixes it can reach to its neighbors. In the emulation, a change on configurations or topology can trigger changes on IP prefixes reachability from certain devices. Such changes propagate among the emulated devices. We state the following lemma:

LEMMA 5.1. *In an emulated BGP network, a boundary is safe if and only if no route update originated in an emulated device passes through the boundary more than once.*

It is straightforward to prove Lemma 5.1, since speaker devices do not need to react if all route updates originated in emulated devices stay within the emulated network, or exit without coming back to the emulated network. Lemma 5.1 applies to all distance vector routing protocols.

Nonetheless, it is hard to apply Lemma 5.1 directly because checking all potential route update paths for all IP prefixes may not be feasible. Hence, we state a stronger version of the lemma, which can be implemented efficiently:

Proposition 5.2. *If the boundary devices of an emulated BGP network are within a single AS and all of the speaker devices are in different ASes, the boundary is safe.*

Proof. If all boundary devices are within a single AS, no route updates can exit the boundary and return again because, to avoid routing loops, BGP does not allow sending route updates back to the same AS. According to Lemma 5.1, the proof completes. □

Note that route updates can return if the devices represented by the speakers (or external) devices arbitrarily modify or remove elements from AS paths, but such cases are rare. In practical production networks, the modifications on AS paths are mostly just repeating individual ASes for multiple times to change the AS path length. In this case, route updates will not return because there will be a loop otherwise.

Proposition 5.2 provides a sufficient condition for checking whether a boundary is safe in BGP network, and so as to search a small safe boundary. For example, the boundary in Figure 7b is safe because S1 and S2 are in a single AS. In addition, we state a even stronger proposition:

Proposition 5.3. *If the boundary devices of an emulated BGP network are in ASes that have no reachability to each other via external networks, the boundary is safe.*

Proof. After a route update is sent out from a boundary device (β) to a speaker device, the route update will never reach any other boundary devices because other boundary devices are either in the same AS as β, or have no connectivity to β. According to Lemma 5.1, the proof completes. □

For instance, assuming operators only want to emulate L1-4 but not T1-4, Figure 7c presents a safe boundary. This is due to boundary devices S1-2, L1-2, L3-4 are in three different ASes that have no reachability to each other without passing the emulated network zone. For example, when link S1-L1 fails, L1 will send withdraw messages for the routes it learns from S1 to T1 and T2, but T1 or T2 will not send the withdraw messages to L2 because L1 and L2 are both in AS200. S1-2 or L3-4 are not affected either, since they are not reachable from T1 or T2. Similar situations happen in T3-4 and L5-6.

Searching safe static boundary in datacenters running BGP. Given a set of input devices, searching for an optimal boundary which satisfies Proposition 5.2 or 5.3 is still difficult in a general network. However, for a Clos-like datacenter network like [4, 5], we derive a useful heuristic based on its special properties: (i) The whole network topology is layered; (ii) Valley routing [9] is now allowed; and (iii) The border switches connected to wide area network (WAN) are on the highest layer and usually share a single AS number.

Our idea is to treat the topology as a multi-root tree with border switches being the roots. Starting from each input device, we add all its parents, grandparents and so on until the border switches into the emulated device set. This is essentially a BFS on a directional graph, which is constructed by replacing the links in the topology with directional child-to-parent edges. Algorithm 1 shows the BFS process. It is easy to verify that the output emulated devices has a safe boundary. We omit the formal proof due to lack of space.

Algorithm 1: FindSafeDCBoundary(G, D)

```
[Input] G: datacenter network topology
[Input] D: input devices by operators to be emulated
[Output] D′: all devices to be emulated
D′ ← ∅;
while $D \neq \emptyset$ do
    d ← D.pop();
    D′.add(d);
    if G.isHighestLayerDevice(d) then
        continue;
    upperDevices ← G.allConnectedUpperLayerDevices(d);
    foreach $dev \in upperDevices$ do
        if $dev \notin D \cup D'$ then
            D.add(dev)
return D′;
```

OSPF networks. OSPF is a link state routing protocol and is widely used in large scale networks as interior gateway protocol (IGP). Unlike BGP, routers in OSPF report the state (e.g., liveness, weights, etc.) of their neighboring links to a database which is located in a designated router (DR) and a backup designated router (BDR). A state change on a link triggers routers attached to the link to report the new link state to DR and BDR. To ensure that validating changes on emulated devices does not require reactions from speaker devices, we state the following:

Proposition 5.4. *If the links between boundary devices and speaker devices remain unchanged in the emulated network, and DR(s) and BDR(s) are emulated devices, the boundary of an OSPF network is safe.*

This is because there is nothing new for speaker devices to report if their links remain unchanged; DR(s) and BDR(s) will always react to changes, since they are always emulated.

According to Proposition 5.4, for a boundary of an OSPF network to be safe, both ends of the links that may suffer changes must be emulated devices. The same conclusion applies on IS-IS protocol.

Software-defined networks (SDN). SDN usually runs distributed routing protocols like BGP or OSPF for the connectivity between the centralized controller and network devices. Propositions 5.2, 5.3 and 5.4 can be used to validate the control network. For the data network, a boundary is safe if it includes all devices whose states may impact the controller's decision. Analyzing the controller's code may generate a tighter condition. We leave it as future work.

6 IMPLEMENTATION

CrystalNet consists of over 10K lines of Python code and includes a few libraries that interact with our internal services and public cloud. We do not customize the switch firmware images we receive from our vendors in any way. In this section, we elaborate some important implementation details.

6.1 Prepare phase

`Prepare` API generates the input for `Mockup`. It consists of generating the topology and configurations, and spawning the VMs. The only input for `Prepare` is a list of device host names that operators want to emulate. Then the orchestrator interacts with internal network management services and clouds to execute the following steps.

Generating topology and configurations. For all devices in the input list, *CrystalNet* identifies the locations in the physical topology and computes a safe boundary. Then *CrystalNet* pulls all related topology, device configurations and routing states snapshots. All the information is then preprocessed and rearranged in a format that `Mockup` can understand. The preprocessing mainly includes adding a unified SSH credentials into the configurations, parsing and reformatting routing states, etc.

VM spawning. *CrystalNet* estimates the number of VMs needed, and spawns them on-demand using cloud APIs. This is key to scalability and reducing the cost. The VMs run a pre-built Linux image that includes all necessary software and supported device containers. Additional images may be pulled during runtime using the Docker engine.

The number and type of VMs needed for the emulation depend on factors. We do not want to spawn too many tiny VMs - this increases the burden on the orchestrator, and can also increase cost. At the same time, we do not want to make each VM too large (and pack a lot of devices on the same VM), since the kernel becomes less efficient in packet forwarding when the virtual interfaces is too high. We have also found that container-based devices typically require more CPU, while VM-based devices require more memory. Finally, *CrystalNet* requires nested VMs (§4.1) for emulating devices VMs rather than containers. Azure supports this option for only certain VM SKUs. Based on these considerations, we typically build emulations out of 4-core 8 or 16GB VMs, although we also use other SKUs under certain conditions.

6.2 Mockup phase

`Mockup` is the core part of *CrystalNet*. The time `Mockup` takes determines the time and cost overhead of running *CrystalNet*. Following the design in §4.1, `Mockup` has two steps. First, it sets up the PhyNet layer and the topology connections. Second, it runs the device software. We aggressively batch and parallelize various operations in the `Mockup` stage. See §8 for performance numbers. Below are the implementation lessons we learned and decisions we made.

Linux bridge or OVS (Open vSwitch)? Both Linux bridge and OVS can forward packets and integrate VXLAN tunnel. While *CrystalNet* supports both, we prefer the former, because we only need "dumb" packet forwarding on the virtual links. The Linux bridge is much faster to setup, especially when *CrystalNet* configures O(1000) tunnels per VM. For efficiency, we also disable *iptables* filtering and Spanning Tree Protocol on bridges.

Running different devices on different groups of VMs. Containers on the same host share the same kernel, which can cause problems. For example, we find that one switch vendor tunes certain kernel settings related to packet checksumming, which can cause collocated devices from other vendors to malfunction. To avoid such problems, *CrystalNet* typically does not instantiate devices from different vendors on the same VM. In short, we create groups of VMs, with each group dedicated to run devices from a particular vendor.

Health check and auto-recovery. VMs may fail or reboot without warning. *CrystalNet* includes a health monitor and repair daemon to recover from such failures. The daemon periodically checks the device uptime, and verifies link status by injecting and capturing packets from both ends. If a problem is found, it alerts the user and clears and restarts the failed VM using the APIs described earlier. Since VMs are independent of one another, other VMs do not need to be restarted or reconfigured.

BGP speaker at the boundary. Our production network relies on BGP routing. Therefore, we surround the emulation boundary with BGP speakers (§5) based on ExaBGP 3.4.17. It can inject arbitrary announcements, dump the received announcements for potential analysis, and does not reflect announcements to other peers.

Integrating P4 ASIC emulator. While the images from the three major vendors come with ASIC emulator [7], the open source switch OS *CTNR-B* does not have one. Therefore, we integrate it with the open source P4 behavior model, *BMv2*, which acts the ASIC emulator and forwards packets.

7 REAL-LIFE EXPERIENCES

CrystalNet has been deployed in our production networks for about 6 months. Operators and engineers use it to validate and de-risk new network designs, major network architecture changes, network firmware/hardware upgrades and network configuration updates. They also use it as a realistic test environment for developing network automation tools, and for developing our in-house switch operating system. In this section, we describe two cases that demonstrate the effectiveness of *CrystalNet* in practice.

Case 1: Migrations to new regional backbones. Our cloud platform allows users to allocate resources at desired locations with a coarse geographical concept of *region*, *e.g.,* east of US, west of Europe. We have multiple data centers (DCs) inside each single region and inter-DC/intra-region traffic used to be carried by an inter-DC wide-area network (WAN). However, as the demands for high capacity and low latency between different DCs within the same region grew, the design was upgraded to include new regional backbone networks that connect DCs in the same region and bypass the WAN to improve performance.

Making such major changes to an operational network is fraught with peril. Operators must guarantee that there is no noticeable disruption of existing traffic in the region during or after the migration. Due to considerations such as hardware capacity limitations, IPv4 address shortages, switch load considerations and security, the routing configurations in the routers that connect data centers to the backbone are already quite complex, and even small mistakes in the migration plan can result in large-scale outages.

CrystalNet de-risked this complex operation by allowing the operators to intensively validate and refine their operational plans and software tools in a high-fidelity emulator.

Using *CrystalNet*, we emulated a network consisting of all spine routers (from *Vendor-A*, container provided) in two large data centers, all routers in the new regional backbones and several core routers in legacy WAN (from *Vendor-B*, VM provided). The boundary was proven to be safe. The configurations, operations plans and software tools are thoroughly tested in this emulated environment. The emulated network required just 150 VMs (the same spec as in Section 6.1).

During testing, operators discovered over 50 bugs in their tools and scripts, some of them could have triggered customer-impacting outages. Such outages can result in large financial penalties for violating SLAs, and more importantly, lead to loss of customer confidence. The final migration plan, perfected on *CrystalNet*, did not trigger any incidents, when carried out in production. There were not even any incidents of casual human errors (e.g. typos etc.), which the operators attributed to intensive practice sessions on the emulator.

Case 2: Switch OS developments. *CrystalNet* also helps our engineers to build a validation pipeline for developing a private version of switch OS *CTNR-B*. We emulate production environments, replace some devices with *CTNR-B*, and verify that there is no change in network behavior. We can even plug in hardware devices running *CTNR-B* into emulated networks for integration tests. Within two months, *CrystalNet* has successfully found O(10) bugs which could have caused catastrophic failures. For instance, failing to update the default route when routes are learned from BGP, failing to forward ARP packets to CPU due to incorrect trap implementation, crashing after several BGP sessions flapped, etc. None of these bugs was found by unit tests or testbed tests in the legacy development pipeline, but they are easy to detect in emulated production environments from *CrystalNet*.

Network	#Borders	#Spines	#Leaves	#ToRs	#Routes
L-DC	O(10)	O(100)	O(1000)	O(3000)	O(20M)
M-DC	O(10)	O(10)	O(100)	O(400)	O(1M)
S-DC	O(1)	O(1)	O(10)	O(100)	O(50K)

Table 3: Datacenter networks used in evaluations. The last column is the total number of routing table entries in all switches.

8 EVALUATION

In this section, we evaluate the performance and cost of *CrystalNet* in emulating production datacenter networks.

8.1 Experiment setup

Networks. We showcase *CrystalNet* for three of our real datacenter networks, as shown in Table 3. *L-DC* is one of our largest datacenter networks, and *M-DC* and *S-DC* are representative median and small size datacenter networks. They all have Clos topology. BGP is used as the only routing protocol. Borders, Spines, Leaves and ToRs are switches on different layers, from the border of datacenter down to connecting servers. ToRs run *CTNR-B*, while Border, Spine and Leaf switches run *CTNR-A*. Both *CTNR-A* and *CTNR-B* are packed into container images.

Configurations and routes. All switches run production configurations, generated by our production network management services. All IP prefixes and routes are taken from a snapshot of the corresponding production networks and injected into emulated networks.

Compute resources. For this evaluation, *CrystalNet* spawns VMs with 4-core and 8GB memory, as described in §6.1. We use different number of VMs for different scales.

Performance metrics. Time is money, especially on clouds. We measure the following time spent on VMs: (i) *Network-ready latency*: the duration from the start of creating an emulation to the moment when all virtual links are up; (ii) *Route-ready latency*: the duration from Network-ready to the moment when all routes are installed and stabilized in all switches. After this, the emulated network is ready for operators to run tests on it. (iii) *Mockup latency*: the sum of network-ready and route-ready latencies. (iv) *Clear latency*: the time required for resetting the VMs to a clean state.

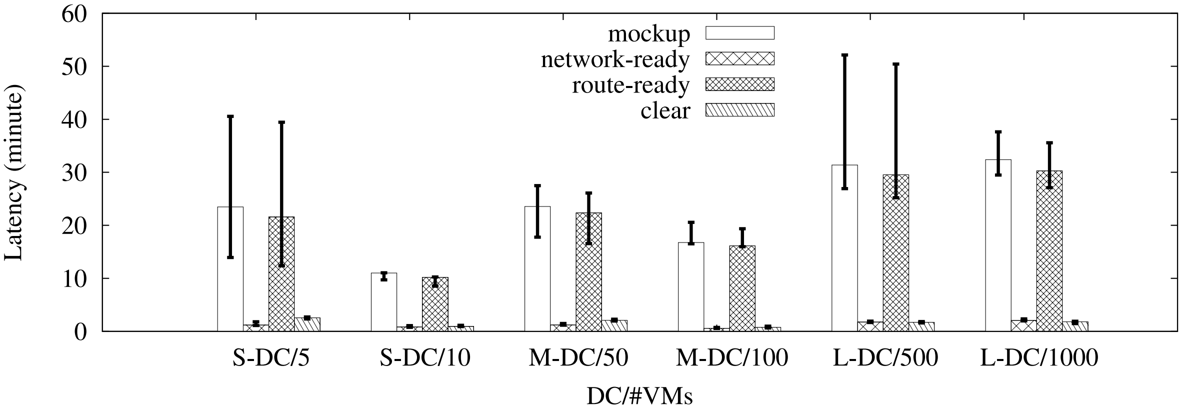
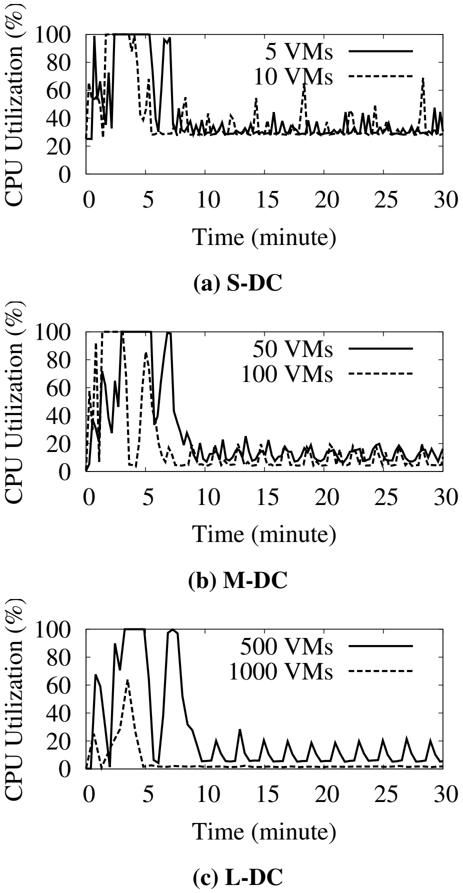


Figure 8: The 10^{th}, 50^{th} and 90^{th} percentile latencies to start/stop emulations with *CrystalNet* for different datacenter scales.



(a) S-DC

(b) M-DC

(c) L-DC

Figure 9: *CrystalNet* VM CPU usage (95^{th} percentile among all VMs).

8.2 Speed of Mockup and Clear operations

***CrystalNet* mockup latency and monetary cost are low for even the largest datacenter.** Figure 8 shows the network-ready, route-ready, and clear latencies of emulating the entire *S-DC*, *M-DC* and *L-DC*. Each setup is repeated 10 times. The median Mockup latency is less than 32 minutes in all cases, and less than 50 minutes at 90th percentile. The clear latency is less than 2 minutes. This is acceptable for validation that is performed on daily basis, *e.g.,* daily rollout or software daily build.

By comparing different VM cluster sizes, we see that more computing resources can achieve lower Mockup latency with smaller variance. However, for large networks, e.g. *L-DC*/500 v.s. *L-DC*/1000, the major bottleneck is the convergence speed of routing algorithms.

The largest datacenter, *L-DC*, requires 500 VMs, which cost ~100 USD per hour. For small networks like *S-DC*, the cost is only ~1 US dollars per hour.

Boot speed of vendor-provided software and routing convergence are the major components of Mockup latency, *CrystalNet* overhead is minimal. Figure 8 shows that the network-ready latency is less than 2 minutes across all datacenter scales and VM cluster sizes. It includes everything described in §4, and only contributes < 5% of total Mockup time.

Route-ready latency is the major component of the Mockup latency. In this phase, all devices setup BGP sessions with their peers and exchange routing information until the routes allover the network converge. Note that route-ready latency depends on network scale, number of routes, the routing protocol and the implementation of vendor-provided software stack, which is not controlled by *CrystalNet*. Figure 9 shows the instantaneous CPU usage of Mockup.[7] In the beginning,

[7] Memory is not the bottleneck (details omitted).

CPU utilization is high, for creating a large number of virtual interfaces and links, and for initializing vendor-provided software. After about 10 minutes, the CPU usage is minimal, but the routes still need 5-20 minutes to converge.

8.3 Speed of recovery from local changes

Two-layer design allows *CrystalNet* to quickly reload individual devices. We compare the time for `Reload` an individual device with *CrystalNet*'s PhyNet-software-separate design and a strawman everything-together design. In *CrystalNet*, the PhyNet layer remains even if device software reboots, so it need not re-create interfaces or links. Therefore, *CrystalNet* takes *3* seconds to reload one device[8] while the strawman takes at least *15* extra seconds for reconfiguring interfaces and links. Some device software do not even support hot plug interfaces (§4.1), and two layer design is thus *required* for such devices.

***CrystalNet* recovers from VM failures quickly.** We also test the time spent on recovering from a failed (and rebooted) VM – i.e. the time required to reset devices and links on that VM. For all the networks we test, the recovery time varies from 10 seconds to 50 seconds, depending on deployment density. This does not include the time needed for the VM to reboot. We are investigating a design where *CrystalNet* keeps a small number of spare VMs in reserve to quickly swap out failed VMs instead of waiting for failed VMs to reboot.

8.4 Importance of safe static boundaries

***CrystalNet* can find a small safe boundary for common validation cases and reduce cost significantly** In above experiments, we always emulate the whole datacenter networks. However, in practice, network operators usually update only a small piece of the network each time. Thus, they only need to validate the operations on that piece. Below we use *L-DC* topology to demonstrate two most common operation cases.

Case-1 : Operators often want to make changes to a single Pod [6], which includes a group of adjacent ToRs and Leaves. As shown in Table 4, the final emulated network found by Algorithm 1 has 4 Leaves and 16 ToRs, which are inside the target Pod, plus additional 64 Spines and 4 Borders. The overall number of emulated devices is 88, which is less than 2% of the whole network.

Case-2 : Operators often want to make changes to the Spine layer of a datacenter network because these devices have sophisticated configurations, like ACLs or route maps. In *L-DC*, to emulate the whole Spine layer, the safe boundary includes the entire Spine layer and Border layer. As shown in Table 4, we need to emulate less than 3% of the devices.

Case	#Borders	#Spines	#Leaves	#ToRs	Proportion
One Pod	4	64	4	16	≤ 2%
All Spines	12	112	0	0	≤ 3%

Table 4: The emulation scales with safe boundaries in *L-DC*.

In both cases, there are also hundreds of speaker devices (not shown in Table 4). However, they are so lightweight that a single VM can support at least 50 of them. Overall, *CrystalNet* required just 20 VMs for the first case, and 30 for the second. The operators were able to perform all necessary tasks and validations on the emulated network. Since emulating the whole network requires upwards of 500 VMs, we conclude that *CrystalNet*'s safe boundary identification feature reduces the cost of running the emulation by over 90%.

9 DISCUSSION

Dealing with non-determinism. *CrystalNet* does not control the order of routing announcements from speaker devices. In most cases, routing protocols like BGP, OSPF are agnostic to the timing and ordering of routing messages [17, 24]. However, when cross-validating the forwarding tables from *CrystalNet* and production, we found some instances of a non-deterministic BGP behavior. This behavior arises when ECMP path selection is used along with IP prefix aggregations. As shown in Figure 1, if R6 chooses path for P3 randomly or basing on timing, the routes for P3 on R6 and R8 will be non-deterministic. We are currently investigating this issue further, since it is nearly impossible to precisely emulate the exact timing and ordering of prefix announcements. In the meantime, we have designed a FIB comparator that accounts for such non-determinism.

Finding the smallest safe boundary for all routing protocols. In §5.2, we provide sufficient conditions to judge whether a given boundary design is safe for specific routing protocols such as BGP, OSPF/IS-IS. These simple guidelines work well in our datacenter networks and inter-datacenter WAN networks. However, it is challenging to design a generic algorithm that can find the smallest safe boundary for arbitrary routing protocols, since the answer depends on not only the routing protocol and network topology but also the specifics of the changes to be made in the emulated network. We leave this as future work.

Limitations of *CrystalNet*. Even though *CrystalNet* allows us to plug in individual hardware devices into the emulated network, it is not our intention to have real hardware make up the majority of the emulated devices. *CrystalNet* is not suitable for catching ASIC problems. It is not suitable for detailed testing of data-plane performance (e.g. bandwidth or latency) – albeit probing packets could be sent for verifying

[8] Stop the container, overwrite configuration files, and restart the container.

routes. *CrystalNet* is not intended to find bugs that arise from slowly accumulating state (e.g. memory leaks) or timing sensitive bugs (*e.g.,* multi-thread racing). These must be found using standard software verification techniques.

Furthermore, *CrystalNet* cannot efficiently answer questions such as "does there exist a scenario in which one or more links could fail, resulting in excessive traffic through switch X?". Brute-force searching such scenarios will require numerous emulations. It may be cheaper to answer such questions using logical verification tools such as Minesweeper [8] – although one must be willing to accept reduced fidelity to use such tools.

Testing methodologies. *CrystalNet* provides a high-fidelity network emulation, and a basic infrastructure to pull state and trace packets through emulated network. We leave it to operators to design testing strategies using these basic hooks to verify whatever it is that they wish to verify. We are actively working to design efficient testing methodologies using *CrystalNet*, including the design of a domain-specific language to specify properties of interest and automatic generation of test cases to verify those properties.

Programmable data planes: *CrystalNet* is also useful to debug and verify the logic in programmable and stateful data plane such as P4 [10]. We are building a flexible debugging and testing environment with *CrystalNet* for future networks with programmable data planes.

10 RELATED WORK

Network emulations. EmuLab [2] and CloudLab [1] provide network emulation services in their own infrastructures. They allow users to define network topologies and capacities and run realistic applications over the emulated networks. However, currently they do not support customized switch firmware or have on-demand scalability since they are limited by the size of their infrastructure. Flexplane [25] is a data plane emulator which is used to test resource management algorithms used in ASICs. It does not target control plane software or configurations and cannot easily scale out on multiple machines. Kang and Tao [20] proposed a framework which leverages containers to stress test the control plane in SDN context. Their focus is on the performance of SDN controllers, while *CrystalNet* can be used in both SDN and traditional distributed networks for correctness. MiniNet [18] (multi-host version) and MaxNet [29] are both container-based network emulators which can run on distributed clusters. However, they are not suitable for emulating large and heterogeneous production networks, because they lack the three key features of *CrystalNet*: first, they do not integrate with multiple public clouds, private clusters and physical devices; second they do not accommodate heterogeneous blackbox device firmwares or provide transparent access to management software and tools used to manage production network, and third, they do not automatically identify safe emulation boundaries, which is necessary to maintain high fidelity at low cost.

Configuration verification. There are multiple efforts [11–13, 15, 23, 30] to use formal verification techniques to check whether configurations in networks meet certain properties. These systems assume an ideal model of device behavior to compute forwarding tables from configuration files. In reality, device behavior is far from ideal (§2). *CrystalNet* is a more realistic way to compute FIBs. The FIBs can be verified using the same verification techniques. Systems like [13] are still useful to model the ideal system behavior and to track provenance. These tools need fewer resources than *CrystalNet*, so network engineers can use them as their first, low-fidelity check before calling upon *CrystalNet*.

Data plane verification. Verifying rules in forwarding tables of network devices [19, 21, 22, 26] is important to systematically detect routing problems such as reachability violations, blackholes and access control failures. As described before, *CrystalNet* facilitates data plane verification by providing forwarding tables from emulations rather than real networks, so that problem can be detected proactively.

11 CONCLUSION

We developed *CrystalNet*, which provides high-fidelity emulations of production environments to validate network operations. *CrystalNet* offers a faithful network control plane, on-demand scalability, uniform management over heterogeneous device software, and safe emulation boundaries. These features are critical for constructing effective tests; and set *CrystalNet* apart from alternatives like testbed experiments, configuration verifications and small-scale network emulators. *CrystalNet* is highly scalable, cost-effective, and in daily use at Microsoft, where it has prevented numerous potential incidences.

12 ACKNOWLEDGMENTS:

We would like to thank many Microsoft engineers who have contributed to the success of this project. These include John Abeln, Dave Carlton, George Chen, Andrew Eaton, Shane Foster, Ivan Lee, Dave Maltz, Haseeb Niaz, Iona Yuan, Changqing Zhu, and many others. We would like to thank our reviewers and our shepherd, Arvind Krishnamurthy, for their helpful comments.

REFERENCES

[1] Cloudlab. https://www.cloudlab.us/.

[2] Emulab. https://www.emulab.net/.

[3] GNS3. https://www.gns3.com/.

[4] Introducing Data Center Fabric, the Next-Generation Facebook Data Center Network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook\-data-center-network/.

[5] Routing Design for Large Scale Datacenters: BGP is a better IGP! https://www.nanog.org/meetings/nanog55/presentations/Monday/Lapukhov.pdf.

[6] Al-Fares, M., Loukissas, A., and Vahdat, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM Computer Communication Review* (2008), vol. 38, ACM, pp. 63–74.

[7] Barefoot. P4 Software Switch. https://github.com/p4lang/behavioral-model/.

[8] Beckett, R., Gupta, A., Mahajan, R., and Walker, D. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 155–168.

[9] Beckett, R., Mahajan, R., Millstein, T., Padhye, J., and Walker, D. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM* (2016), ACM, pp. 328–341.

[10] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[11] Fayaz, S. K., Sharma, T., Fogel, A., Mahajan, R., Millstein, T., Sekar, V., and Varghese, G. Efficient Network Reachability Analysis using a Succinct Control Plane Representation. In *OSDI* (2016), USENIX Association, pp. 217–232.

[12] Feamster, N., and Balakrishnan, H. Verifying the Correctness of Wide-Area Internet Routing.

[13] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., and Millstein, T. D. A General Approach to Network Configuration Analysis. In *NSDI* (2015), pp. 469–483.

[14] Ford, B., Srisuresh, P., and Kegel, D. Peer-to-Peer Communication Across Network Address Translators. In *ATC* (2005), pp. 179–192.

[15] Gember-Jacobson, A., Viswanathan, R., Akella, A., and Mahajan, R. Fast Control Plane Analysis using an Abstract Representation. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 300–313.

[16] Google. Google Compute Engine Incident NO.16007. Connectivity issues in all regions. https://status.cloud.google.com/incident/compute/16007.

[17] Griffin, T. G., Shepherd, F. B., and Wilfong, G. The Stable Paths Problem and Interdomain Routing. *IEEE/ACM Transactions on Networking (ToN) 10*, 2 (2002), 232–243.

[18] Handigol, N., Heller, B., Jeyakumar, V., Lantz, B., and McKeown, N. Reproducible Network Experiments using Container-Based Emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 253–264.

[19] Horn, A., Kheradmand, A., and Prasad, M. R. Delta-net: Real-time Network Verification Using Atoms. *arXiv preprint arXiv:1702.07375* (2017).

[20] Kang, H., and Tao, S. Container-based emulation of network control plane. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems* (2017), ACM, pp. 24–29.

[21] Kazemian, P., Varghese, G., and McKeown, N. Header Space Analysis: Static Checking for Networks. In *NSDI* (2012), vol. 12, pp. 113–126.

[22] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P. Veriflow: Verifying Network-Wide Invariants in Real Time. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 467–472.

[23] Lopes, N. P., Bjørner, N., Godefroid, P., Jayaraman, K., and Varghese, G. Checking Beliefs in Dynamic Networks. In *NSDI* (2015), pp. 499–512.

[24] Moy, J. T. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley Professional, 1998.

[25] Ousterhout, A., Perry, J., Balakrishnan, H., and Lapukhov, P. Flexplane: An experimentation platform for resource management in datacenters. In *NSDI* (2017), pp. 438–451.

[26] Plotkin, G. D., Bjørner, N., Lopes, N. P., Rybalchenko, A., and Varghese, G. Scaling Network Verification using Symmetry and Surgery. In *POPL* (2016).

[27] Premji, A., Lapukhov, P., and Mitchell, J. RFC 7938: Use of BGP for Routing in Large-Scale Data Centers, 2016.

[28] Sung, Y.-W. E., Tie, X., Wong, S. H., and Zeng, H. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference* (2016), ACM, pp. 426–439.

[29] Wette, P., Draxler, M., Schwabe, A., Wallaschek, F., Zahraee, M. H., and Karl, H. Maxinet: Distributed Emulation of Software-Defined Networks. In *Networking Conference, 2014 IFIP* (2014), IEEE, pp. 1–9.

[30] Yuan, L., Chen, H., Mai, J., Chuah, C.-N., Su, Z., and Mohapatra, P. Fireman: A Toolkit for Firewall Modeling and Analysis. In *Security and Privacy, 2006 IEEE Symposium on* (2006), IEEE, pp. 15–pp.

[31] Zhai, E., Chen, R., Wolinsky, D. I., and Ford, B. Heading Off Correlated Failures through Independence-as-a-Service. In *OSDI* (2014), pp. 317–334.

[32] Zhu, Y., Kang, N., Cao, J., Greenberg, A., Lu, G., Mahajan, R., Maltz, D., Yuan, L., Zhang, M., Zhao, B. Y., et al. Packet-Level Telemetry in Large Datacenter Networks. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 479–491.